2026-08-19 · 11 MIN READ

Three ways to smuggle SQLite into Nix

The core of nixpkgs-multiverse (/2026/08/09/nixpkgs-multiverse-every-version-that-ever-existed), when you strip away the Nix API and the CLI, is an index. It is a map from `(attribute, version)` to the revision that shipped it as a JSON file.

> 1 There are actually a few other files that drive other features such as the statistics or "fast mode" (/2026/08/14/nixpkgs-multiverse-fast-mode), but they are all JSON as well.

```
$ ls -lh index/
-rw-r--r--. 1 fmzakari fmzakari 7.5M Aug 19 13:57 history.json
-rw-r--r--. 1 fmzakari fmzakari 5.3M Aug 19 13:57 versions.json
```

As of 9cc0209 (https://github.com/fzakaria/nixpkgs-multiverse/commit/9cc02098e177f784f822c57973ebfc3c02c21bed), `versions.json` is 5.3 MiB and `history.json` is 7.5MiB covering 305,492 package versions across 31,904 packages and 1,534 revisions.

The Nix API loads the JSON files lazily and are all read via `builtins.fromJSON`:

NIX

```nix
index = builtins.fromJSON (builtins.readFile ./index/versions.json);
```

I would like to enrich the data with even more information however it comes at a cost: mo'data, mo'problems.

The goal of the project is to minimize the number of Nixpkgs that are downloaded. If we merely swap fetching huge Nixpkgs for huge JSON, it's not a clear win.

For now we have to be judicious about what we store in the JSON files and think of clever encoding schemes to make the data small and compact.

If we were not constrained to the Nix `builtins`, we would leverage established technologies to efficiently encode our dataset that allow multiple query access patterns: databases!

Let's say we were not restricted to JSON, do we have any other options?

§ One lookup costs the whole file

Why are large JSON files so problematic? `builtins.fromJSON` is *eager*. There is no lazy JSON in Nix, no streaming parse (i.e. "just give me this one key"). The moment you touch the result you have parsed all 5.3 MB and materialised all 305,492 values on the Nix heap.

In the case of the multiverse, asking for one package costs the same as what asking for all of them.

> NOTE
>
> The lookup itself is not the problem. Nix attribute sets are a sorted array, so access is a binary search, not a scan. The cost is entirely in the JSON parse and in allocating the values and downloading a large file.

If we want to do alternate questions over the index, we have to make sure we keep the answers efficiently stored to better match the access pattern.

What we want is obvious. We want a way to efficiently encode the data and a declarative way to define queries: we want SQLite!

2 nixpkgs-multiverse (https://github.com/fzakaria/nixpkgs-multiverse) already exports a SQLite database as a package to help others explore this data.

```
$ sqlite3 index.db "SELECT version, rev FROM versions WHERE attr='hello'"
2.10|728
...
0.01s, 4 MB
```

Nix *by default* cannot do this. Unfortunately there is no `builtins.sqlite`, although I think there should be…

Turns out though there are knobs we can touch or sources we can patch to get what we want anyways, albeit each one has a caveat. 😈

§ One: `builtins.exec`

I was surprised I did not know about this `builtin`, and it has been around since release 1.11.9 (https://github.com/NixOS/nix/issues/1300) in April 2017. It is the ultimate escape hatch for a variety of use-cases when you simply can't get them done with what's available.

`builtins.exec` takes a list of strings, runs the program, and **parses its stdout as a Nix expression**.

It is gated behind a setting that makes it clear it's unsafe.

```
$ nix eval --option allow-unsafe-native-code-during-evaluation true \
    --expr 'builtins.exec [ "/bin/sh" "-c" "echo 42" ]'
42
```

For integration, SQLite is perfectly capable of printing the Nix syntax. We never need a serialisation format in between as we make SQLite emit the attrset directly:

NIX

```nix
let
  versionsOf = attr: builtins.exec [
    "${sqlite}/bin/sqlite3" "-noheader" "-separator" "" "./index.db"
    ''
      SELECT '{' || group_concat(
               '"' || version || '" = ' ||
               COALESCE(CAST(rev AS TEXT), 'null') || ';', ' ')
           || '}'
      FROM versions WHERE attr = '${attr}';
    ''
  ];
in
  versionsOf "hello"
```

```
$ nix eval --impure -f query.nix \
           --option allow-unsafe-native-code-during-evaluation true
{
  "2.10" = 728; "2.12" = 822; "2.12.1" = 1369;
  "2.12.2" = 1486; "2.12.3" = null; "2.7" = 0; "2.8" = 13;
}
```

The caveat is that every query is now a `fork`, an `exec`, a process image of SQLite, and a re-parse of the output through the Nix parser. If you do not plan to execute many queries that overhead is likely acceptable given the simplicity of the integration.

§ Two: `builtins.importNative`

From researching `builtins.exec`, I stumbled upon `builtins.importNative`. It takes a path to a shared object and a symbol name, `dlopen`s it, and calls that symbol. It landed in **1.8**, December 2014.

> 3 The C++ field was originally called `enableImportNative` and was renamed to `enableNativeCode` for `exec`.

The shared object must implement the following signature:

CPP
```
extern "C" typedef void (*ValueInitializer)(EvalState & state, Value & v);
```

We can define a new *native* function that returns the versions for our input:

CPP
```
extern "C" void nix_sqlite_versions(EvalState & state, Value & v)
{
    v.mkPrimOp(new PrimOp{
        .name = "nix_sqlite_versions",
        .args = {"dbPath", "attr"},
        .arity = 2,
        .impl = versions,
    });
}
```

The implementation is ordinary C++ using the Nix API. Below is a snippet of the implementation, making sure to cache our `sqlite3` handles to avoid the same

startup penalty as `builtins.exec`:

```cpp
/* The whole point: the database handle outlives a single query, so the
   b-tree pages we touch stay warm for the rest of the evaluation. */
std::map<std::string, sqlite3 *> handles;

void versions(EvalState & state, const PosIdx pos,
              Value ** args, Value & v)
{
    std::string path(state.forceStringNoCtx(*args[0], pos, "..."));
    std::string attr(state.forceStringNoCtx(*args[1], pos, "..."));

    // cached across calls
    auto * db = openOnce(state, pos, path);

    sqlite3_stmt * stmt = nullptr;
    sqlite3_prepare_v2(db,
                       "SELECT version, rev "
                       "FROM versions "
                       "WHERE attr = ?1",
                       -1, &stmt, nullptr);
    sqlite3_bind_text(stmt, 1, attr.data(),
                      attr.size(), SQLITE_TRANSIENT);

    /* ... collect rows ... */

    /* Build the attrset directly. No text ever exists. */
    auto bindings = state.buildBindings(rows.size());
    for (auto & [version, rev] : rows) {
        auto & slot = bindings.alloc(state.symbols.create(version));
        if (rev) slot.mkInt(*rev); else slot.mkNull();
    }
    v.mkAttrs(bindings);
}
```

Using it looks like this:

```
$ nix eval --impure \
    --option allow-unsafe-native-code-during-evaluation true \
    --expr '(builtins.importNative
                  ./libnixsqlite.so "nix_sqlite_versions"
            ) "./index.db" "hello"'
{
  "2.10" = 728; "2.12" = 822; "2.12.1" = 1369;
  "2.12.2" = 1486; "2.12.3" = null; "2.7" = 0; "2.8" = 13;
}
```

§ Three: a giant Nix file

This section was added after publishing based on an idea from rickynils (https://github.com/rickynils).

Nix is often described as resembling JSON and there is a very easy translation from JSON to Nix. What if instead of reading JSON we read the same contents but as a `.nix` file?

Theoretically it should have no parser boundary, no `fromJSON`, and no serialisation format at all. The index becomes an expression the evaluator already knows how to read.

The idea would be to leverage Nix's laziness. Nix attribute set values are thunks, so in principle you should be able to `import` a very large expression, touch one attribute, and never pay for instantiating the rest.

Transforming the index is a dozen lines of Python, and produces something very similar to the JSON:

NIX

```nix
{
  revisionCount = 1534;
  attrs = {
    "2048-in-terminal" = { "2015-01-15" = 157; "2017-11-29" = 166; };
    "2bwm" = { "0.2" = 166; };
    "389-ds-base" = { "1.3.3.9" = 14; "1.3.5.15" = 100; "1.3.5.19" = 166;
    # ... 31,901 more
  };
}
```

6.0 MiB of Nix, against 5.3 MiB of JSON, holding identical data.

```
$ nix eval --impure --expr '(import ./index.nix).attrs.hello'
{
  "2.10" = 728; "2.12" = 822; "2.12.1" = 1369;
  "2.12.2" = 1486; "2.12.3" = null; "2.7" = 0; "2.8" = 13;
}
```

§ Four: `builtins.wasm`

Determinate Systems shipped another option (https://determinate.systems/blog/builtins-wasm/) in March of 2026: `builtins.wasm`, which calls a function inside a WebAssembly module.

> 4 Eelco gave a talk about this at SCALE 23x (https://www.socallinuxexpo.org/scale/23x/presentations/builtinswasm-nix-meets-webassembly).

The motivation was similar to wanting to extend Nix surface area but avoid expanding `builtins`. Wasm is sandboxed and deterministic, so unlike the two builtins above, the goal is to provide a *safe escape-hatch*.

WebAssembly is a binary instruction format for a stack-based virtual machine. The claim is that it is well suited for Nix because it has *deterministic execution*, which is a lot more restrained than a backdoor `builtins.exec`.

§ Writing a module

A module needs to export `memory`, an initialiser called `nix_wasm_init_v1`, and the entry point.

```rust
#![no_std]
#![no_main]
type ValueId = u32;

#[panic_handler]
fn panic(_: &core::panic::PanicInfo) -> ! {
    core::arch::wasm32::unreachable()
}

// Host functions supplied by the Nix evaluator.
#[link(wasm_import_module = "env")]
unsafe extern "C" {
    fn get_int(v: ValueId) -> i64;
    fn make_int(n: i64) -> ValueId;
}

#[unsafe(no_mangle)]
pub extern "C" fn nix_wasm_init_v1() {}

fn fib(n: i64) -> i64 {
    if n <= 1 { 1 } else { fib(n - 1) + fib(n - 2) }
}

#[unsafe(no_mangle)]
pub extern "C" fn fib_entry(arg: ValueId) -> ValueId {
    unsafe { make_int(fib(get_int(arg))) }
}
```

Nixpkgs already includes the target for cross-compilation, so making one is pretty straightforward:

```nix
pkgs.runCommand "nix-wasm-rust-fib"
{
  nativeBuildInputs = [ pkgs.rustc pkgs.lld ];
  src = ./modules.rs;
} ''
  mkdir -p $out
  rustc --target wasm32-unknown-unknown --crate-type cdylib -O \
    -o $out/modules.wasm $src
''
```

```
$ nix eval --extra-experimental-features wasm-builtin \
      --expr 'builtins.wasm { path = ./modules.wasm;
                              function = "fib_entry"; } 30'
1346269
```

You call back into the evaluator through the Nix API functions, so a wasm module builds real Nix values, similar to `builtins.importNative` minus the footgun.

§ Can I haz SQLite?

SQLite ships an official wasm build (https://sqlite.org/wasm/doc/trunk/index.md), so the pieces seem to be sitting right there and the gears in my mind began to turn.

Initial attempts to try and load a SQLite database with the traditional Nix `builtins` were a bit of a failure as Nix strings cannot contain NULL bytes.

```
$ nix eval --impure --expr 'builtins.stringLength (builtins.readFile ./ind
error: the contents of the file '/tmp/mvsql/index.db' cannot be represente
```

Thankfully, with the help of some additional due-diligence by LLMs, we discovered that one of the Nix API functions is not in the blog post:

```
CPP
```

```cpp
/**
 * Read the contents of a file into Wasm memory. This is like calling
 * `builtins.readFile`, except that it can handle binary files that
 * cannot be represented as Nix strings.
 */
uint32_t read_file(ValueId pathId, uint32_t ptr, uint32_t len)
```

`read_file` is *specifically* designed for this problem. This function allows a WASM module to pull arbitrary raw-bytes off disk into its memory.

Unfortunately, it's a little *too broad* in that it reads **the complete file** which is kind of overkill and what we are trying to avoid from our initial JSON solution.

In the pursuit of exploration, let's *patch* the implementation and augment the API to allow random access and partial read of a file. Turns out the patch to add is relatively small and straightforward.

```
/**
 * Read a range of a file into Wasm memory, starting at `offset`
 * and copying at most `len` bytes.
 * Returns the number of bytes actually copied.
 */
uint32_t read_file_range(ValueId pathId, uint64_t offset,
                         uint32_t ptr, uint32_t len)
{
    auto & pathValue = getValue(pathId);
    auto path = state.realisePath(noPos, pathValue);

    auto buf = memory().subspan(ptr, len);

    /* If this is a real file on disk, do a positional read*/
    if (auto physical = path.getPhysicalPath()) {
        AutoCloseFD fd{open(physical->string().c_str(),
                            O_RDONLY | O_CLOEXEC)};
        if (!fd)
            throw SysError("opening file '%s'", physical->string());
        auto n = pread(fd.get(), buf.data(), len, offset);
        if (n < 0)
            throw SysError("reading file '%s'", physical->string());
        return n;
    }

    /* Otherwise fall back to materialising the whole file. */
    auto contents = path.readFile();
    if (offset >= contents.size())
        return 0;
    auto n = std::min<size_t>(len, contents.size() - offset);
    memcpy(buf.data(), contents.data() + offset, n);
    return n;
}
```

Now we have everything we need to hook up SQLite and a custom virtual filesystem (VFS) layer to read from the provided `/nix/store` path entry.

We build a WASM target of SQLite and we set `SQLITE_OS_OTHER=1`. That flag removes SQLite's entire VFS layer and requires us to supply one.

```nix
pkgs.pkgsCross.wasi32.stdenv.mkDerivation {
  pname = "sqlite-nix-wasm";
  buildPhase = ''
    $CC -O2 -o sqlite_nix.wasm \
      -I${amalgamation} ${amalgamation}/sqlite3.c sqlite_nix.c \
      -DSQLITE_OS_OTHER=1 \
      -DSQLITE_THREADSAFE=0 \
      -DSQLITE_OMIT_LOAD_EXTENSION \
      -DSQLITE_OMIT_WAL \
      -Wl,--export-memory
  '';
}
```

We provide the build a simple implementation of the `xRead` API which is a call-back into the Nix evaluator via that newly exposed `nix_read_file_range` function. Everything else is stubs.

```c
static const sqlite3_io_methods nixIoMethods = {
  .iVersion               = 1,
  .xClose                 = nixClose,
  .xRead                  = nixRead,
  .xFileSize              = nixFileSize,
  .xDeviceCharacteristics = nixDeviceCharacteristics,
  /* ... the rest are stubs ... */
};

static int nixRead(sqlite3_file *f, void *buf,
                   int amt, sqlite3_int64 off)
{
  NixFile *p = (NixFile *) f;
  /* The one line that matters: SQLite's pager asks
     for a page, and we ask the Nix evaluator for
     exactly those bytes. */
  unsigned got = nix_read_file_range(p->pathId, (unsigned long long) off,
                                     buf, (unsigned) amt);

  if (got < (unsigned) amt) {
    memset((char *) buf + got, 0, (unsigned) amt - got);
    return SQLITE_IOERR_SHORT_READ;
  }
  return SQLITE_OK;
}
```

> NOTE
>
> Unfortunately `builtins.wasm` gives every call a **fresh instance**. This is deliberate from the implementation, meaning we pay some startup code each time although not quite as drastic as a `fork` & `exec`

The `sqlite_nix` WASM module takes an attrset of `{ db, sql }` and returns one attrset per row.

[5] The full `sqlite_nix.c`, the VFS, the build derivation and the Nix patch are all in this gist (https://gist.github.com/fzakaria/8fe754ee7db752a24cb9c55b38492844).

We can provide it any arbitrary SQL and now query our dataset!

NIX

```nix
# query.nix
builtins.wasm { path = ./sqlite_nix.wasm; } {
  db  = ./index.db;
  sql = "SELECT version, rev FROM versions
         WHERE attr = 'hello' ORDER BY version";
}
```

```
$ nix eval --extra-experimental-features wasm-builtin -f query.nix
[ { rev = 728; version = "2.10"; } { rev = 822; version = "2.12"; }
  { rev = 1369; version = "2.12.1"; } { rev = 1486; version = "2.12.2"; }
  { rev = null; version = "2.12.3"; } { rev = 0; version = "2.7"; }
  { rev = 13; version = "2.8"; } ]
```

The benefit of SQL is that now we are not limited to the shape of the data in JSON.

NIX

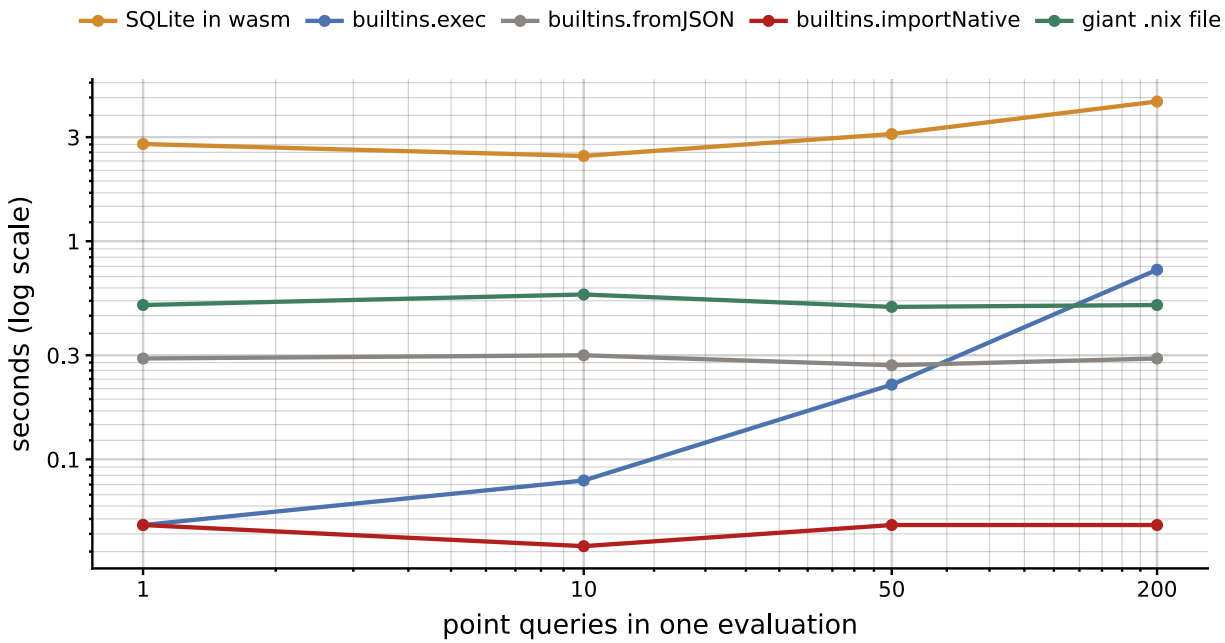
```nix
# which packages have shipped the most versions?
sql = "SELECT attr, COUNT(*) AS versions FROM versions
       GROUP BY attr ORDER BY versions DESC LIMIT 3";
# => [ { attr = "linux"; versions = 548; }
#      { attr = "linux_latest"; versions = 540; }
#      { attr = "freefall"; versions = 534; } ]
```

That is a real full SQLite with all the bells and whistles: query planner, aggregates and subqueries, b-tree descent through an index, executing inside the Nix evaluator. All through WebAssembly. 🤯

Every one of those answers is byte-identical to what the `sqlite3` CLI gives for the same query.

§ Benchmark

How do these compare? Here is every approach answering the same question: "which revisions shipped this package?" either against the same 22 MB SQLite build of the index or the whole-file JSON/Nix equivalent.



(/assets/plotnine/b8febc306a46ca88.svg)

As we initially complained, `fromJSON` **is a flat line in the wrong place.** It is 0.29s whether you ask one question or two hundred, because the 5.3 MB parse happens once and dominates everything after it.

The giant `.nix` file is the same flat line, drawn higher. It is roughly twice the time and 1.7× the memory of the JSON it replaced. Surprisngly, laziness never gets a chance to help: importing the file and touching *nothing at all* already costs 0.53s. The baseline cost is the parse, and the parse is eager as we well. Turns out parsing Nix expressions is even more expensive than JSON. Nix has run the file through its Bison grammar, build an AST for all 305,492 entries, and add every attribute name into the symbol table. `fromJSON` skips the AST entirely and goes straight from bytes to values, which is why the format with a "serialisation boundary" beats the one without.

`builtins.exec` **starts the cheapest and climbs**, roughly 3.8 ms per query of `fork` + `exec` + Nix-parsing the output. It crosses `fromJSON` somewhere around eighty queries.

`builtins.importNative` **is flat and nearly free**, 0.05s across the whole range since we *reuse SQLite instantiations* across multiple invocations. The database is opened once for the entire evaluation and the pages stay warm.

Unfortunately, **SQLite in wasm is dominated by a fixed cost**, roughly **2.5 s** before the first query, then about **7 ms** each query thereafter. That 2.5 s is Cranelift compiling 1.1 MB of SQLite. Right now that is a limitation of the WASM implementation however Eelco has mentioned that the generated code could be cached on disk in the future across invocations.

For a lock file pinning thirty packages, `fromJSON` still wins outright at the current index size.

§ What I actually want

None of these is right for shipping the multiverse index, and I am not going to make `nixpkgs-multiverse` depend on `allow-unsafe-native-code-during-evaluation`. Asking people to run their evaluator with native code loading enabled so my flake can be faster is not a worthwhile request *at the moment*.

For now, the index stays JSON and I'm holding back on some of the more loftier ideas I have that require *a lot more data*.

Although philosophically I only use CppNix (https://github.com/nixos/nix), I was a little intrigued and impressed with what the ecosystem could unlock with WASM. There are definitely some warts however such as waiting for it to JIT and the developer-experience of maybe having checked-in compiled blobs but there is definitely potential to unlock a variety of problems.

/nix/store/mbq2amkjjl24h2qp4aj21a2x9ld30kad-fzakaria.com